2INC0 - Operating Systems

Condition synchronization

Geoffrey Nelissen

IRiS
Interconnected Resource-aware Intelligent Systems

TU/e Technische Universiteit Eindhoven University of Technology

Where innovation starts

Announcements

- **Check the additional exercises on Canvas** (in modules)
- New programming **workshop** next Friday
  (using threads, mutexes, semaphores, …)

Course Overview

- **Introduction to operating systems** (lecture 1)
- **Processes, threads and scheduli**
- **Concurrency and synchronizatio**
  - atomicity and interference (lecture 4)
  - actions synchronization (lecture 5)
  - **condition synchronization (lecture 6)**
  - deadlock (lecture 7)
- **File Systems** (lecture 8)
- **Memory management** (lectures 9+10)
- **Input/output** (lecture 11)

- Dangers of concurrency (**race conditions**)
- How to prove properties using **traces**
- Synchronization (**critical sections**)

- Check complex properties using **topology invariants**
- Use semaphores to **synchronize actions** and enforce new properties

- Enforce properties that **cannot be checked just by counting** actions

Agenda

- Condition variables
- Monitors
  - Signaling disciplines

Two complementary tools for condition synchronization

Actions synchronization

Invariant

- An assertion that **holds at all control points** of a program

Topology invariant

- An invariant **derived** directly **from the program** code
- Examples:
  - number of times an **operation is executed** in comparison to another action.
  - **Value of a variable** based on the number of times a set of actions was executed

Actions synchronization

- Uses **semaphores** to **synchronizes actions** in different tasks to **enforce new invariants** (called synchronization invariants or synchronization conditions)
- Steps:
  - Name actions that may influence the synchronization condition
  - Write the synchronization condition as a function of actions counts in the form $\sum_i a_i \times cA_i \leq \sum_j b_j \times cB_j + e$ for non-negative constants a_i, b_j, and e .
  - introduce **semaphore** ***s***, with **initial value** $s_0 = e$
  - **replace action** A_i **with** $P(s)^{a_i}\ A_i$
  - **replace action** B_j **with** $B_j\ V(s)^{b_j}$

Topology invariant and concurrency

```
int x := 7;
int y := 1;
int z := 6;
```

Go to menti.com and use 2438 8084

```
Task1 = |[
    while (true) {
        X1: x:=x+6;
        Y1: y:=2*y;
        if( x < y ){
            Z1: z++;
        }
    }
]|
```

||

```
Task2 = |[
    while (true) {
        Y2: y:=y-2;
        X2: x--;
        if( x > 8 ) {
            Z2: z--;
        }
    }
]|
```

cX = number of times action *X* executed

$x = ?$

Topology invariant and concurrency

TU/e

```
int x := 7;
int y := 1;
int z := 6;
```

```
Task1 = |[
   while (true) {
      X1: x:=x+6;
      Y1: y:=2*y;
      if( x < y ){
         Z1: z++;
      }
   }
]|
```

||

```
Task2 = |[
   while (true) {
      Y2: y:=y-2;
      X2: x--;
      if( x > 8 ) {
         Z2: z--;
      }
   }
]|
```

cX = number of times action *X* executed

$\cancel{x = 7 + 6 \times cX_1 - cX_2}$

Is it really correct?

No, because X1 and X2 are **not atomic**.
We must first protect them against **race conditions**

IRiS

Topology invariant

```
int x := 7;
int y := 1;
int z := 6;
```

```
Task1 = |[
    while (true) {
        X1: <x:=x+6>;
        Y1: < y:=2*y>;
        if( x < y ){
            Z1: < z++>;
        }
    }
]|
```

||

```
Task2 = |[
    while (true) {
        Y2: < y:=y-2>;
        X2: < x-->;
        if( x > 8 ) {
            Z2: < z-->;
        }
    }
]|
```

cX = number of times action *X* executed

Now assume all actions are atomic:

$$x = 7 + 6 \times cX_1 - cX_2$$

$$z = ?$$

IRiS

Topology invariant

```
int x := 7;
int y := 1;
int z := 6;
```

```
Task1 = |[
    while (true) {
        X1: <x:=x+6>;
        Y1: < y:=2*y>;
        if( x < y ){
          Z1: < z++>;
        }
    }
]|
```

||

```
Task2 = |[
    while (true) {
        Y2: < y:=y-2>;
        X2: < x-->;
        if( x > 8 ) {
          Z2: < z-->;
        }
    }
]|
```

cX = number of times action *X* executed

Now assume all actions are atomic:

$$x = 7 + 6 \times cX_1 - cX_2$$

$$z = 6 + cZ_1 - cZ_2$$

Actions synchronization: what can we enforce?

```
int x := 7;
int y := 1;
int z := 6;
```

Go to menti.com and use 2438 8084

```
Task_1 = |[
    while (true) {
        X_1: x:=x+6;
        Y_1: y:=2*y;
        if( x < y ){
            Z_1: z++;
        }
    }
]|
```

||

```
Task_2 = |[
    while (true) {
        Y_2: y:=y-2;
        X_2: x--;
        if( x > 8 ) {
            Z_2: z--;
        }
    }
]|
```

cX = number of times action *X* executed

Which of the **synchronization invariants** presented to you **can be enforced with actions synchronization**?

Mentimeter answers

- $x \geq 4$
- $y > 0$
- The execution repeats the exact sequence X1 Y2 X2 Y1
- $z < x$
- $x \geq 4$ and $z < x$
- $y < x$
- Y1 executes at least twice as often as Z2
- $x \geq 4$ or $z < x$

Value of *y* cannot be derived from the number of times we execute Y1 and Y2 because of the multiplication in Y1

Using action synchronization can only check and enforce one of the two condition, and will thus not allow for minimal waiting

Green = can be enforced with actions synchronization

Red = cannot be enforced with actions synchronization

Condition synchronization

- **Condition synchronization**
  - **explicit communication (signalling)** between tasks
  - when just **counting is not enough** to solve a synchronization problem
  - or to **simplify** otherwise **complex sequences of semaphore operations**, e.g., *P(a); P(a); P(b); P(m); ....V(a); V(b);*

Two principles:

Where a condition may be violated: **check** and potentially **wait**
Where a condition *may have become true*: **signal** waiters

Condition synchronization: building blocks

- ***ConditionVar*** *cv ;* (often **associated with** a condition: **a boolean expression *B* in terms of program variables;**)

- 4 basic operations on variable *cv*:
  - *Wait(…, cv)* — **suspend** execution of caller
  - *Signal(cv)* — **free *one* task** suspended on *Wait(cv)* (void if there is none)
  - *Sigall(cv)* — **free *all* tasks** suspended on *Wait(cv)*
  - *Empty(cv)* — Check if "there is no task suspended on *Wait(cv)*" (returns true or false)

Example

Enforce the synchronization invariant $x \geq 0$

```
int x := 0;
```

```
Task T1
|[
   while( true ){
         z:=z+1;
         x := x + 50;
         x := 2*x;
   }
]|
```

||

```
Task T2
|[
   while( true ){
         x := x-10;
         y := x+3;
   }
]|
```

Step 1: identify critical sections and protect them

Example

```
mutex m := 1;
int x := 0;
```

Enforce the synchronization invariant $x \geq 0$

```
Task T1
|[
  while( true ){
      z:=z+1;
      lock(m);
      x := x + 50;
      x := 2*x;
      unlock(m);
  }
]|
```

||

```
Task T2
|[
  while( true ){
      lock(m);
      x := x-10;
      unlock(m);
      y := x+3;
  }
]|
```

Step 1: identify critical sections and protect them

Example

```
mutex m := 1;
int x := 0;
```

Enforce the synchronization invariant $x \geq 0$

```
Task T1
|[
  while( true ){
      z:=z+1;
      lock(m);
      x := x + 50;
      x := 2*x;
      unlock(m);
  }
]|
```

||

```
Task T2
|[
  while( true ){
      lock(m);
      x := x-10;
      unlock(m);
      y := x+3;
  }
]|
```

Step 2: identify where a condition may become false and add a guard

Example

```
ConditionVar cv;
 mutex m := 1;
int x := 0;
```

Enforce the synchronization invariant $x \geq 0$

```
Task T1
|[
  while( true ){
      z:=z+1;
      lock(m);
      x := x + 50;
      x := 2*x;
      unlock(m);
  }
]|
```

||

```
Task T2
|[
  while( true ){
      lock(m);
      while( x<10 ){
          Wait( m, cv);
      }
      x := x-10;
      unlock(m);
      y := x+3;
  }
]|
```

Step 2: identify where a condition may become false and add a guard

Example

```
ConditionVar cv;
 mutex m := 1;
int x := 0;
```

Enforce the synchronization invariant $x \geq 0$

```
Task T1
|[
  while( true ){
      z:=z+1;
      lock(m);
      x := x + 50;
      x := 2*x;
      unlock(m);
  }
]|
```

||

```
Task T2
|[
  while( true ){
      lock(m);
      while( x<10 ){
          Wait( m, cv);
      }
      {x>=10}
      x := x-10;
      unlock(m);
      y := x+3;
  }
]|
```

Step 3: identify where a condition may become true and signal

Example

Enforce the synchronization invariant $x \geq 0$

```
ConditionVar cv;
 mutex m := 1;
int x := 0;
```

```
Task T1
|[
  while( true ){
       z:=z+1;
       lock(m);
       x := x + 50;
       x := 2*x;
       Sigall(cv);
       unlock(m);
  }
]|
```

||

```
Task T2
|[
  while( true ){
       lock(m);
       while( x<10 ){
          Wait( m, cv);
       }
       {x>=10}
       x := x-10;
       unlock(m);
       y := x+3;
  }
]|
```

Wait(m,cv) is the short-hand notation for ***<unlock(m); wait(cv);> lock(m);***

Must be **atomic**!
Why?

Example

```
ConditionVar cv;
 mutex m := 1;
int x := 0;
```

Enforce the synchronization invariant $x \geq 0$

```
Task T1
|[
  while( true ){
      z:=z+1;
      lock(m);
      x := x + 50;
      x := 2*x;
      Sigall(cv);
      unlock(m);
  }
]|
```

||

```
Task T2
|[
  while( true ){
      lock(m);
      while( x<10 ){
          Wait( m, cv);
      }
      {x>=10}
      x := x-10;
      unlock(m);
      y := x+3;
  }
]|
```

Why do we use ***Sigall(.)*** instead of *Signal(.)*?

Wait(m,cv) is the short-hand notation for ***<unlock(m); wait(cv);> lock(m);***

Must be **atomic**! Otherwise, another task may **signal before** T2 **starts to wait** and T2 will **never wakeup**

Example

Enforce the **synchronization invariant** $x \geq 0$

```
ConditionVar cv;
 mutex m := 1;
int x := 0;
```

```
Task T1
|[
  while( true ){
       z:=z+1;
       lock(m);
       x := x + 50;
       x := 2*x;
       Sigall(cv);
       unlock(m);
  }
]|
```

||

```
Task T2
|[
  while( true ){
       lock(m);
       while( x<10 ){
            Wait( m, cv);
       }
       {x>=10}
       x := x-10;
       unlock(m);
       y := x+3;
  }
]|
```

How do we decide **what to write as a guard condition**?

Wait(m,cv) is the short-hand notation for ***<unlock(m); wait(cv);> lock(m);***

Why do we use ***Sigall(.)*** instead of *Signal(.)*? **More than one instance** of T2 **could execute their critical section** for each execution of T1's critical section

Must be **atomic**! Otherwise, another task may **signal before** T2 **starts to wait** and T2 will **never wakeup**

What to write as a guard condition?

1. **Take the synchronization invariant** to maintain, i.e.,
   $x \geq 0$
2. **Compute preconditions** for the **statements that might disturb the invariant**, *i.e.*,
   x := x-10

   ➔ Substitute disturbing statement in the synchronization invariant

   $$x - 10 \geq 0 \Leftrightarrow x \geq 10$$
3. **Negate** the resulting condition, i.e.,
   !(x ≥ 10)
   or equivalently
   (x<10)

```
Task T2
|[
  while( true ){
      lock(m);
      while( x<10 )
          Wait( m, cv);
      }
      x := x-10;
      unlock(m);
      y := x+3;
  }
]|
```

Exercise (typical exam question)

```
int f := 5;
int w := 0;
```

```
Task T =
|[ while( true ){
        transport();
        f := f+1;
        w := w+3;
}
]|
```

||

```
Task P =
|[ while( true ){
        w := w-2;
        produce();
        f := f+1;
}
]|
```

||

```
Task S =
|[ while( true ){
        f := f-2;
        transport();
        w := w+3;
}
]|
```

Question:
Enforce the following synchronization invariants using condition variables

- **I1:** $w \geq 0$
- **I2:** $f \geq 2 * w$

Prevent race conditions on *f* and *w*
Functions *produce()* and *transport()* cannot be in critical sections

Exercise (typical exam question)

Go to menti.com and use
2438 8084

```
int f := 5;
int w := 0;
mutex m := 1;
ConditionVar cv1, cv2;
```

```
Task T =
|[ while( true ){
        transport();

        lock(m);
        f := f+1;
        sigall(cv2);
        unlock(m);

        lock(m);
        while(…)
           wait(m, cv2);
        w := w+3;
        sigall(cv1);
        unlock(m);
}
]|
```

||

```
Task P =
|[ while( true ){
        lock(m);
        while(…)
           wait(m, cv1);
        w := w-2;
        sigall(cv2);
        unlock(m);

        produce();

        lock(m);
        f := f+1;
        sigall(cv2);
        unlock(m);
}
]|
```

What should be the guard condition?

||

```
sk S =
hile( true ){
        lock(m);
        while(…)
           wait(m, cv2);
        f := f-2;
        unlock(m);

        transport();

        lock(m);
        while(…)
           wait(m, cv2);
        w := w+3;
        sigall(cv1);
        unlock(m);
}
]|
```

Question:
Enforce the following synchronization invariants using condition variables

- **I1:** $w \geq 0$
- **I2:** $f \geq 2 * w$

Prevent race conditions on *f* and *w*
Functions *produce()* and *transport()* cannot be in critical sections

Exercise (typical exam question)

Go to menti.com and use 2438 8084

```
int f := 5;
int w := 0;
mutex m := 1;
ConditionVar cv1, cv2;
```

```
Task T =
|[ while( true ){
        transport();

        lock(m);
        f := f+1;
        sigall(cv2);
        unlock(m);

        lock(m);
        while(…)
           wait(m, cv2);
        w := w+3;
        sigall(cv1);
        unlock(m);
}
]|
```

What should be the guard condition?

```
Task P =
|[ while( true ){
        lock(m);
        while(w<2)
           wait(m, cv1);
        w := w-2;
        sigall(cv2);
        unlock(m);

        produce();

        lock(m);
        f := f+1;
        sigall(cv2);
        unlock(m);
}
]|
```

||

```
Task S =
|[ while( true ){
        lock(m);
        while(…)
           wait(m, cv2);
        f := f-2;
        unlock(m);

        transport();

        lock(m);
        while(…)
           wait(m, cv2);
        w := w+3;
        sigall(cv1);
        unlock(m);
}
]|
```

Question:
Enforce the following synchronization invariants using condition variables

- **I1:** $w \geq 0$
- **I2:** $f \geq 2 * w$

Prevent race conditions on *f* and *w*
Functions *produce()* and *transport()* cannot be in critical sections

Exercise (typical exam question)

```
int f := 5;
int w := 0;
mutex m := 1;
ConditionVar cv1, cv2;
```

```
Task T =
|[ while( true ){
        transport();

        lock(m);
        f := f+1;
        sigall(cv2);
        unlock(m);

        lock(m);
        while( w>(f-6)/2 )
           wait(m, cv2);
        w := w+3;
        sigall(cv1);
        unlock(m);
}
]|
```

||

```
Task P =
|[ while( true ){
        lock(m);
        while(w<2)
           wait(m, cv1);
        w := w-2;
        sigall(cv2);
        unlock(m);

        produce();

        lock(m);
        f := f+1;
        sigall(cv2);
        unlock(m);
}
]|
```

||

```
Task S =
|[ while( true ){
        lock(m);
        while(…)
           wait(m, cv2);
        f := f-2;
        unlock(m);

        transport();

        lock(m);
        while(…)
           wait(m, cv2);
        w := w+3;
        sigall(cv1);
        unlock(m);
}
]|
```

Question:

Enforce the following synchronization invariants using condition variables

- **I1:** $w \geq 0$
- **I2:** $f \geq 2 * w$

Prevent race conditions on *f* and *w*

Functions *produce()* and *transport()* cannot be in critical sections

Exercise (typical exam question)

```
int f := 5;
int w := 0;
mutex m := 1;
ConditionVar cv1, cv2;
```

```
Task T =
|[ while( true ){
        transport();

        lock(m);
        f := f+1;
        sigall(cv2);
        unlock(m);

        lock(m);
        while( w>(f-6)/2 )
           wait(m, cv2);
        w := w+3;
        sigall(cv1);
        unlock(m);
}
]|
```

||

```
Task P =
|[ while( true ){
        lock(m);
        while(w<2)
           wait(m, cv1);
        w := w-2;
        sigall(cv2);
        unlock(m);

        produce();

        lock(m);
        f := f+1;
        sigall(cv2);
        unlock(m);
}
]|
```

||

```
Task S =
|[ while( true ){
        lock(m);
        while(f<2*w+2)
           wait(m, cv2);
        f := f-2;
        unlock(m);

        transport();

        lock(m);
        while(w>(f-6)/2)
           wait(m, cv2);
        w := w+3;
        sigall(cv1);
        unlock(m);
}
]|
```

Question:
Enforce the following synchronization invariants using condition variables

- **I1:** $w \geq 0$
- **I2:** $f \geq 2 * w$

Prevent race conditions on *f* and *w*
Functions *produce()* and *transport()* cannot be in critical sections

Signalling strategies

- Two signalling strategies
  - ***Signal*** wakes up **one** waiter on *cv;*
  - ***Sigall*** wakes up **all** waiters on *cv.*

- Which one to choose: *Signal* or *Sigall ?*
  - Requires a careful analysis to ensure both **correctness** and **efficiency**.
  - ***Sigall*** is **always correct**, but **often inefficient.**

- ***Wait(m, cv)*** can also be **extended with** a **timeout** mechanism
  - To recheck a condition even when there is no signal
  - May increase robustness against (programming) errors
    (e.g., signal **not sent**, or use of ***signal()* instead of *sigall()*** in the wrong situation, normally or abnormally **terminated signaller**, …)

POSIX: condition variables (1003.1c)

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
status = pthread_cond_init (&cond, attr);              /* should return 0              */
status = pthread_cond_destroy (&cond);                 /* idem                         */
status = pthread_cond_wait (&cond, m);                 /* semaphore m is associated with
                                                        * all critical sections        */
status = pthread_cond_timedwait (&cond, m, exp); /*  exp: max. waiting time; returns
                                                        * ETIMEDOUT after exp.*/
status = pthread_cond_signal (&cond);                  /* signal one waiter            */
status = pthread_cond_broadcast (&cond);               /* signal all waiters           */
```

PYTHON: condition objects

```
m = threading.Lock()
cv = threading.Condition(lock=m)
```

```
cv.acquire()
while not condition_respected():
    cv.wait()
execute_critical_section()
cv.release()
```

```
cv.acquire()
do_something()
cv.notify()
cv.release()
```

PYTHON: condition objects (alternative)

```
cv = threading.Condition()
```

```
with cv:
    cv.wait_for(condition_respected)
    execute_critical_section()
```

```
with cv:
    do_something()
    cv.notify()
```

Agenda

- Condition variables
- Monitors
    - Signaling disciplines

Monitors

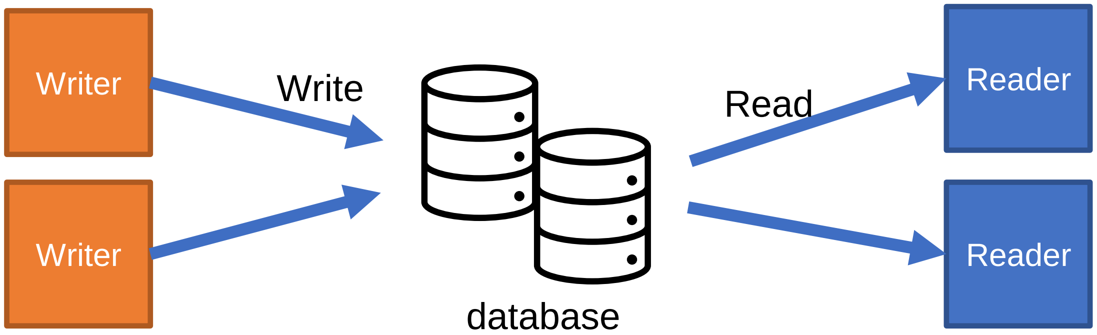
- A **monitor** is an object that contains a combination of data and operations on those data
  - **Data** record the **state of the monitor**
  - **Procedures** of a monitor are executed **under mutual exclusion**
    - **At most one task** can be **"inside the monitor"**
  - **Synchronization** is implemented using ***condition variables***

- Often used as a **programming pattern** rather than natively supported by OS and programming languages

> - Implemented in **Java** using the keywords *synchronize*, *wait*, and *notify*
> - Supported by **Ada** with ***protected objects***

Programming language for **safety-critical systems** (used in **avionics** and **space**)

Example: readers-writers problem



Why using **mutexes** or semaphores is **not a good solution**?

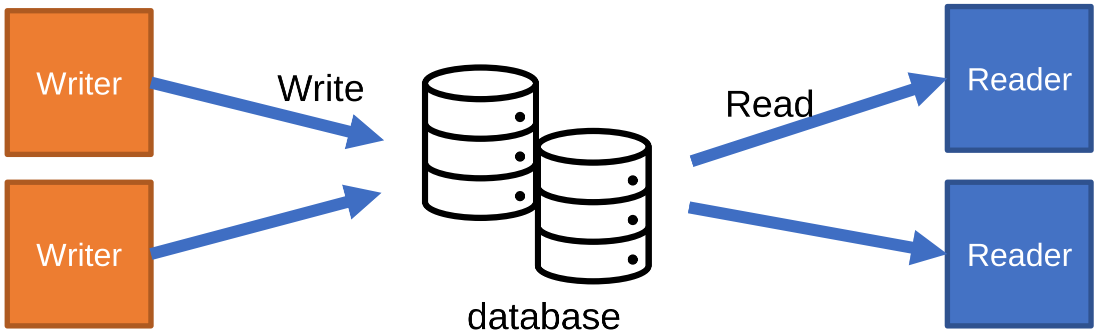
At any given time, we can have **several readers but no writer**, or **one writer and no readers**

Proc *Writer* =
|[
  while(*true*){
    Write_action();
  }
]|

Proc *Reader* =
|[
  while(*true*){
    Read_action();
  }
]|

Example: readers-writers problem

Synchronization invariant to satisfy
$(nr = 0 \wedge nw \leq 1) \vee nw = 0$



At any given time, we can have **several readers but no writer**, or **one writer and no readers**

We use a **monitor** to **synchronize accesses** to the read and write actions

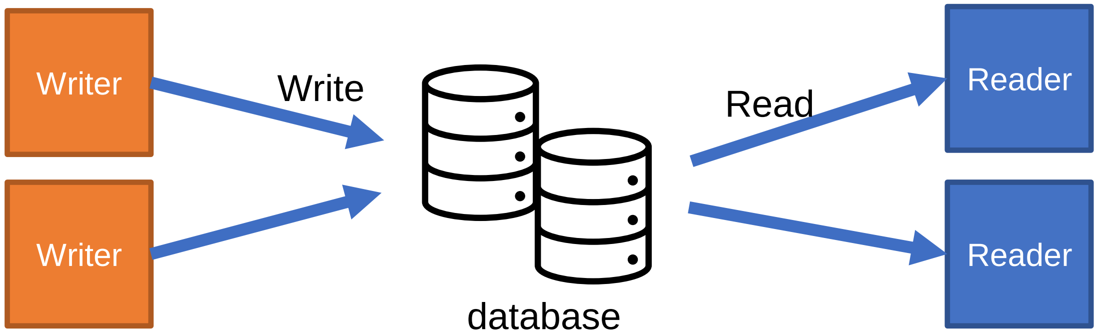
Proc *Writer* =
|[
  while(*true*){
    Write_action();
  }
]|

Monitor *MonReadWrite* =
|[
  int *nr* := *0*;   *// number of readers reading*
  int *nw* := *0*; *// number of writers writing*
]|

Proc *Reader* =
|[
  while(*true*){
    Read_action();
  }
]|

Example: readers-writers problem

Synchronization invariant to satisfy
$(nr = 0 \wedge nw \leq 1) \vee nw = 0$



MonReadWrite *RW;*

Proc *Writer* =
|[
  while(*true*){
    RW.writeEntry();
    Write_action();
    RW.writeExit()
  }
]|

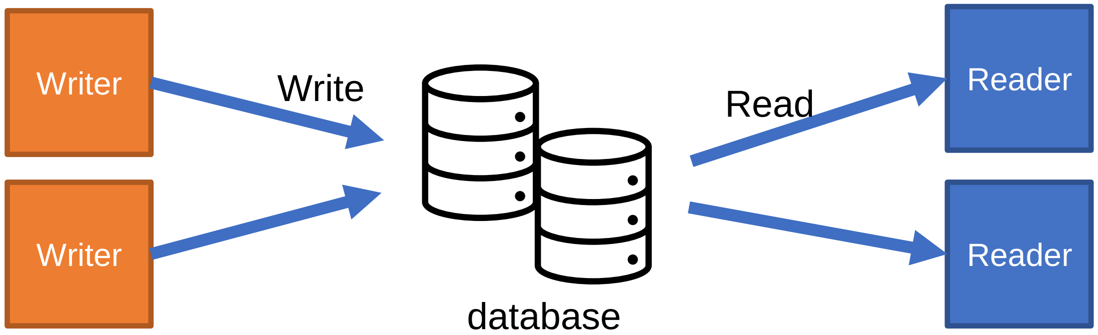
Proc *Reader* =
|[
  while(*true*){
    RW.readEntry();
    Read_action();
    RW.readExit();
  }
]|

Monitor *MonReadWrite* =
|[
  int *nr := 0;* *// number of readers reading*
  int *nw := 0; // number of writers writing*
  Proc writeEntry() { *nw++;* }
  Proc writeExit() { *nw--;* }
  Proc readEntry() { *nr++;* }
  Proc readExit() { *nr--;* }
]|

Add **entry and exit protocols** around the read and write actions

Example: readers-writers problem

Synchronization invariant to satisfy
$(nr = 0 \wedge nw \leq 1) \vee nw = 0$




```
MonReadWrite RW;
```

```
Proc Writer =
|[
  while( true ){
     RW.writeEntry();
     Write_action();
     RW.writeExit()
  }
]|
```

```
Proc Reader =
|[
  while( true ){
     RW.readEntry();
     Read_action();
     RW.readExit();
  }
]|
```

```
Monitor MonReadWrite =
|[
  ConditionVar readCond, writeCond;
  int nr := 0;   // number of readers reading
  int nw := 0; // number of writers writing
  Proc writeEntry() {
     while(nw>0 or nr>0) wait(writeCond);
     nw++;
  }
  Proc writeExit() {
     nw--; signal(writeCond); sigall(readCond);
  }
  Proc readEntry() {
     while(nw>0) wait(readCond);
     r++;
  }
  Proc readExit() {
     nr--; signal(writeCond);
  }
]|
```

Note 2: we use **signal for write** but **sigall for read**

Why?

Note 1: we do not use a mutex because monitors already enforce mutual exclusion between their procedures

Example: readers-writers problem

Synchronization invariant to satisfy
$(nr = 0 \wedge nw \leq 1) \vee nw = 0$

TU/e

Why did we **not implement the *Write_action()* and *Read_action()*** procedures in the monitor, instead of using *Entry* and *Exit* procedures?

Would limit the number of readers reading concurrently to one because the monitor procedures are mutually exclusive.

```
MonReadWrite RW;
```

```
Proc Writer =
|[
  while( true ){
      RW.writeEntry();
      Write_action();
      RW.writeExit()
  }
]|
```

```
Proc Reader =
|[
  while( true ){
      RW.readEntry();
      Read_action();
      RW.readExit();
  }
]|
```

```
Monitor MonReadWrite =
|[
   ConditionVar readCond, writeCond;
   int nr := 0;   // number of readers reading
   int nw := 0; // number of writers writing
   Proc writeEntry() {
      while(nw>0 or nr>0) wait(writeCond);
      nw++;
   }
   Proc writeExit() {
      nw--; signal(writeCond); sigall(readCond);
   }
   Proc readEntry() {
      while(nw>0) wait(readCond);
      r++;
   }
   Proc readExit() {
      nr--; signal(writeCond);
   }
]|
```

IRiS

Implementing a monitor in Java

```
class MonReadWrite {
  private int nr = 0;   // number of readers reading
  private int nw = 0; // number of writers writing

  public synchronized void writeEntry() {
    while(nw>0 or nr>0) {
      try{ wait(); }
      catch(InterruptedException e){}
      finally{}
    }
    nw++;
  }
  public synchronized void writeExit() {
    nw--; notifyAll();
  }
  public synchronized void readEntry() {
    while(nw>0) {
      try{ wait(); }
      catch(InterruptedException e){}
      finally{}
    }
    nr++;
  }
  public synchronized void readExit() {
    nr--; notifyAll();
  }
}
```

Java has a **single implicit condition variable** associated with every object

Agenda

- Condition variables
- Monitors
  - Signaling disciplines

Structuring by monitors

- A monitor:
  - **encapsulates** relevant **shared variables**
  - provides well-defined **operations on the shared variables**

- A monitor **provides exclusion**.
  - **At most one task may be inside** the monitor at any given time.

- **Question**: ***Which task is inside the monitor right after the signal?***
  - The answer depends on the *signalling discipline (scheduling discipline)*.

Scheduling/signalling disciplines

- The discipline defines what happens to the monitor upon a *signal*.

- Four disciplines are used in various implementations
  - signal & **exit**
  - signal & **continue**
  - signal & **wait**
  - signal & **urgent wait**

- *Correctness of solution depends on signalling discipline!*

Signal and Exit

- **The task executing a signal exits** the monitor immediately.
  - Monitor access is given to a waiter on that variable, if any.
  - ***Sigall*** **is not supported**

- **Signalling strategy:**
  - **upon leaving the monitor**, a task performs **at most one signal** on a **condition variable *cv* that is non-empty**…
  - … and for which **the corresponding condition *B(cv)* holds.**

- **Property: The condition *B(cv)* certainly holds after *Wait().***

Signal and Continue

- **The task executing a signal continues to use the monitor**, until a *Wait* or until the end of the routine.
  - Any **task released** by this signal **compete** with the other tasks to obtain monitor access again.

- **Property: The condition may or may not hold after *Wait*.**

- **Signalling strategy:**
  - during execution of a monitor routine **all relevant conditions are signalled**.
  - **after each wait()**, the condition is **evaluated** again.

Signal and Wait

- **The task executing a signal is stopped in favour of the signalled process.**
  - The **signaller waits** to access the monitor again after the *Signal*, …
  - … **and competes** with all other tasks **again**.

- **Signalling strategy:** similar to *Signal & Exit*.

- **Property: The condition holds after *Wait*.**

Signal and Urgent Wait

- **The task executing a signal is stopped in favour of the signalled process.**
  - The **signaller waits** to access the monitor again after the *Signal*, …
  - … **but does not compete** with all other tasks again.
  - **Signaller has priority** over other tasks.
- **Signalling strategy:** similar to *Signal & Exit*.
- **Property: The condition holds after *Wait*.**

Summary

- **Limitations of actions synchronization using semaphores**
  - **Can only enforce invariants based on action counts**
- **How to synchronize tasks using condition variables**
  - **Using guards and signals**
- **Presented the notion of monitors**
  - **Object made of procedures and private data**
  - **Mutual exclusion on the execution of procedures (only one task in the monitor)**
  - **Condition variables can be used to enforce invariants on states reachable by the monitor**

- **Next week,** we discuss how to **detect, prevent** and **recover from deadlocks** in more details
- **I released additional exercises on condition synchronization to train yourself**